Investigation into Improving Convergence in Steepest-Descent Training Methods for Artificial Neural Networks

Student: Josh Rushton, University of Idaho
Advisor: John M. Neuberger, Northern Arizona University

Preface

This report describes my exploration of the field of Artificial Neural Networks during the NSF's 1998 Research Experiences for Undergraduates Program at Northern Arizona University.

As one might expect, this project involved quite a bit of background reading, as well as some efforts pursuing avenues whose relevance turned out to be less clear than expected. Generally, I have included these topics in this report for three reasons: first, for the sake of simple completeness; second, the reader may naturally be inclined to pursue the same directions and may find an outline of the virtues and problems of our implementations of these inclinations helpful; finally, I find most of the results interesting in their own rite. With these considerations in mind, hopefully the reader will find the organization of this report somewhat sensible.

--Josh Rushton

Section 1: Introduction

1.1 What is a Neural Network?

In studying artificial neural networks, it is always important to bear in mind the role of the biological model of neural systems. While the structure and performance of organic neural systems is obviously a rich source of inspiration and motivation for the field of ANNs, the actual implementation of an ANN uses components quite different from their biological counterparts[1]. For example, a unit operation in the brain is much slower than that of a silicon logic gate, but the brain is far more energy efficient in terms of unit operations. A difference more pertinent to our interest in the brain's model is the massive parallelism of the brain, a feature we currently cannot mimic on a large scale with linear computing techniques (i.e., processing a sequence of bits one at a time.) Naturally, then, in a survey of the field of ANNs we expect to see a strong biological influence, but not a perfectly true incarnation of biological structures.

By *neural network* (artificial by default,) we mean a collection of *nodes*, or neurons, along with a set of directed, weighted connections among the nodes. Some of the nodes are designated *input nodes* and some are *output nodes*. Associated with each node is an activation function--activation functions are typically uniform throughout the network, but they may vary if the network's designer so chooses. The weighted connections are analogous to synapses, and likewise the activation function attempts to model the firing of a neuron.

Let's outline the activation of a neural network from start to finish. Since there may be multiple input nodes, the input to the network is configured as a vector, referred to as an *input vector*,where each component of the vector is the input to a corresponding input node. Each input node applies its activation function to the node's input. The output of each node is the result of this function application.

Now, for any node j in the network which has a connection from an input node, the sum of the products $y_i w_{ji}$, where y_i is the output of input neuron i and w_{ji} is the weight of the connection *to j from i*, is given to the activation function of neuron j. The next set of calculations is performed similarly within neurons which have weighted connections from the neurons j_1, j_2, ..., where the j_i are the neurons with connections from one or more input nodes. This process, called the *forward pass*, continues until the only activated neurons are the output neurons. The ordered set of outputs from all the output neurons is called the *output vector*.

For the sake of simplicity, we will deal with more structured neural nets than the general one described above. In particular,we will look at *feed-forward layered ANNs*. By *feed-forward*, it is meant that there are no cycles in the network. For example, if there is a connection from node j to node i, then there must not be any path from node i back to j. *Layered* means the nodes are arranged into disjoint sets, or layers, which we enumerate 0,1,2,...,N for an N-layer network, where layer 0 is the input layer, and layer N is the output layer. In such a network, for $1 \leqslant I \leqslant N$, each node in layer I receives input only from nodes in layer I - 1. Naturally, the output of layer 0 is the input vector, and the output from layer N is the output vector. (See Figure1 in word_diagrams.)

▪ 1.2 Activation Functions

In keeping with the biological model, activation functions for the neurons are typically chosen to exhibit an "all or none" behavior. That is, for arguments below a certain value, the output may be, say, zero and for outputs above that value, the output may be, say, one. Some of the more common functions which have (or approximate) this property are:

Figure 1

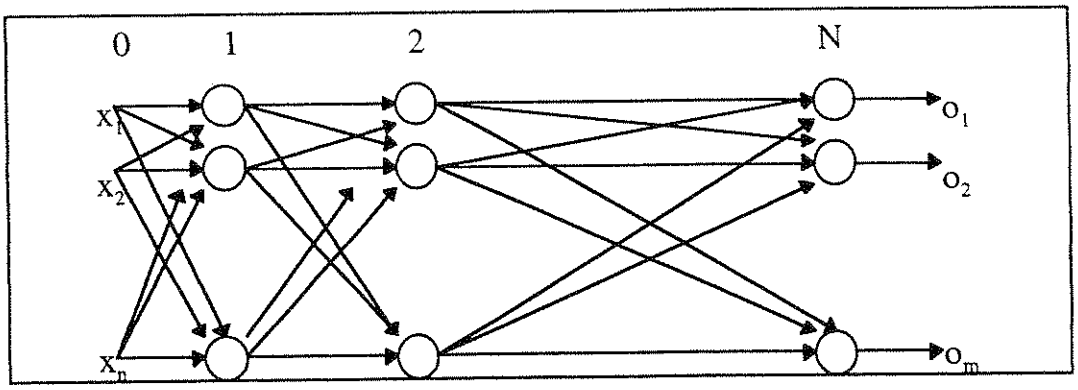


Figure 2

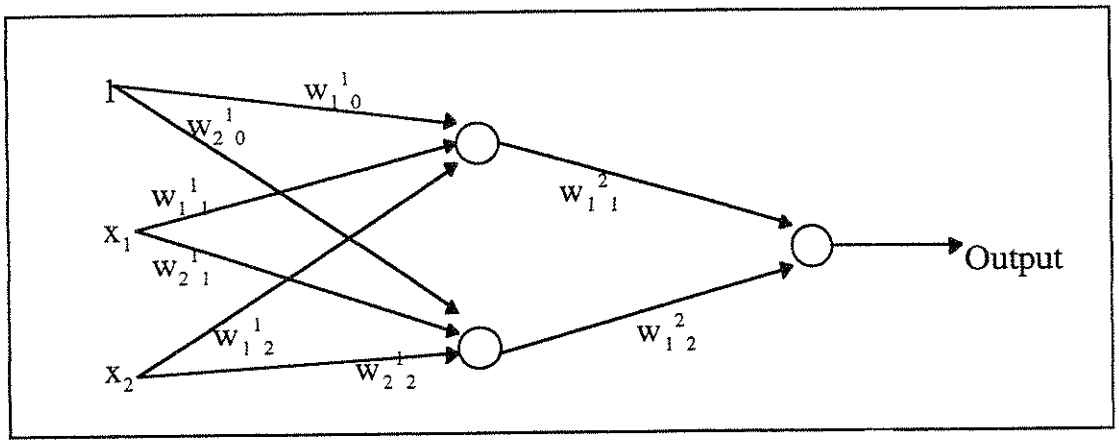
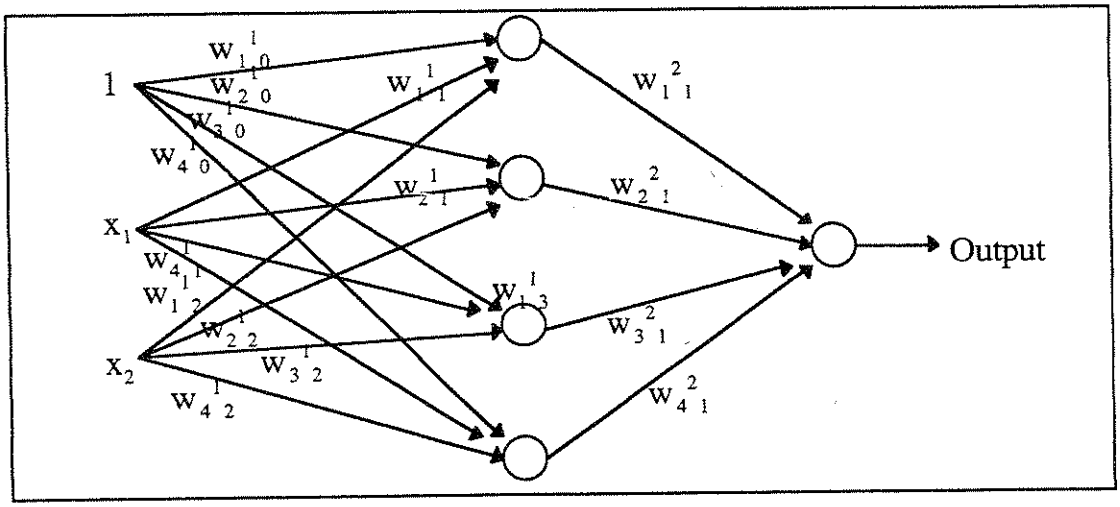
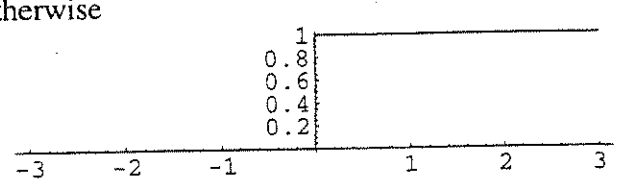


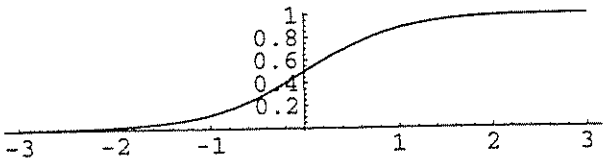
Figure 3



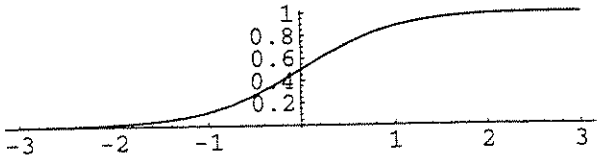
$$\varphi(x) = \begin{cases} +1 & \text{for } x > 0 \\ 0 & \text{otherwise} \end{cases}$$



$\varphi(x) = \frac{1}{2}[1 + \tanh(x)]$



$\varphi(x) = 1/(1 + e^{-ax})$, for some constant a. In the graph below, a = 2.



While the first of these functions is most true to the biological model, the second two have properties of smoothness, strict increase, and asymptotic behavior which make them more popular in ANN implementations. We refer to such functions as sigmoids[1], and all of the programs in this report use the latter activation function, $\varphi(x) = 1/(1 + e^{-ax})$.

▪ 1.3 Training an ANN

Once one has decided to implement an ANN to solve a problem, the first thing to be done is to determine an *architecture* for the network. The *architecture* of a network is the set of nodes and the existence of interconnections among the nodes of the network, as well as the designations of *input nodes* and *output nodes*. The *values* of the weights of the interconnections are not considered part of the architecture. For the process of training a network, we will consider the architecture fixed and try to

adjust the free parameters (e.g., the weights) so as to minimize the network's error.

To do this, we need a precise definition of error. For many problems, we can design a set of input vectors which is representative of the problem in that a network which responds correctly to each of these input vectors stands a good chance of responding correctly to any case of the problem (*generalizing*.) We must next determine the correct output vectors corresponding to these input vectors (ANNs are often applied to problems whose solutions are known, the utility of the network lying in its ability to automate responses, as opposed to coming up with original solutions.) The pair of one of these input vectors and its desired output vector is called a *training pair*, and the set of all training pairs is the *training set*. Algorithms which involve the use of a known training set are called *supervised learning methods*. Let $(\boldsymbol{x}_i,\boldsymbol{d}_i)$ be the i^{th} training pair, and let $F(\boldsymbol{x}_i,\boldsymbol{w}_k) = \boldsymbol{o}_i$ be the output vector generated by the network from the input vector $\boldsymbol{x}_i$,with the weights $\boldsymbol{w}_k$. The reason for the subscript on **w** will be made clear in the next section. The function notation is intentional: since for any input vector, the network will generate exactly one output vector, the operation of the network can be thought of as a function. Finally, we can define the error due to the i^{th} training vector as $\boldsymbol{e}_i = \boldsymbol{d}_i - \boldsymbol{o}_i$, and the total error of the network over the training set is $E = \frac{1}{2} \sum_{i=1}^{N} \|\boldsymbol{e}_i\|^2$. If the architecture and training set are both fixed, then this error is a function of the weights, so it makes sense to talk about adjusting the weights to minimize the error.

There are many algorithms available for minimizing the error in this sense, each with its own set of merits and shortcomings. The algorithm described in the next section is the most popular algorithm in current usage for supervised learning in multilayer ANNs [1] and is the focus of the bulk of this paper.

■ Section 2: Steepest Descent and Back-propagation

■ 2.1 Fundamentals

Suppose we have a value y = f(**x**) for a function f and vector **x**, and that we wish to minimize y. A simple iterative method for reducing y (and hopefully eventually minimizing it) is to take an initial guess $y_0 = f(\boldsymbol{x}_0)$, and then let $\boldsymbol{x}_{n+1} = \boldsymbol{x}_0 - \mu \nabla_E f(\boldsymbol{x}_n)$, where μ is a usually small positive real number, referred to as the *step size*, and $\nabla_E f$ is the standard Euclidean gradient of f. If $||\nabla_E f|| > 0$ and μ is small enough, then the inequality $f(\boldsymbol{x}_n) > f(\boldsymbol{x}_{n+1})$ will hold. Furthermore, the *direction* $\nabla_E f(\boldsymbol{x}_n)$ is the direction in which small changes in **x** result in the most rapid decrease in f(**x**). Hence, the method is called *steepest descent*.

Recall from the last section that for a fixed architecture and training set, we can think of the error of a network as a function of the weights of the network. In fact, if we so desired, we could write out this function explicitly. Unfortunately, even for relatively simple networks this expression would be quite complicated. If we are to apply the steepest descent method to the error as a function of the weights, we might be inclined to write out this function and differentiate it with respect to each of the weights to compute the gradient, but in short order we would see what a daunting task this is. We might also notice patterns among the gradient components and a great deal of repetition of subexpressions. In his 1974 dissertation, Paul Werbos developed a method which takes full advantage of this repetition[1]. The method involves the computation of a recursively defined quantity which yields the gradient components of the error function. The expression of gradient components in terms of this recursive definition is commonly referred to as the *generalized delta rule* (GDR), and *back-propagation* is the algorithm which computes the gradient via the GDR.

In my reading about the derivation of the GDR in [1], [2], and [3], I was unable to find an

explicit statement that back-propagation *is* steepest descent. In order to be certain, as well as to develop my own understanding, I set out to prove this fact. (I am not claiming this has not been proven before, but rather that the applied approach of the textbooks finds no use in including such a proof.)

▪ 2.2 Proof of Equivalence of Generalized Delta Rule and Steepest Descent

The following will be our notation:

$y_i^L \equiv$ the output from neuron i of layer L;

$v_i^L \equiv$ the sum of the inputs to neuron i of layer L;

$\varphi_i^L \equiv$ the sigmoid associated with neuron i of layer L;

$w_{ji}^L \equiv$ the weight of the connection to neuron j of layer L from i of layer L-1;

[Note the following relationships:

$y_i^L = \varphi_i^L(v_i^L)$

$v_j^L = \sum_{i=1}^{I} w_{ij}^L \, y_j^{L-1}$, where I is the number of nodes in layer L-1.]

$e_k \equiv d_k - y_k^N$, where N is the index of the output layer;

$E \equiv \frac{1}{2} \sum_{i=1}^{I} (e_i)^2$, where there are I nodes in the output layer.

[We will evaluate the gradient of this quantity with respect to the weights, even though E refers only to the error due to a single training vector. One might expect a steepest descent traversal of the average error surface over all training pairs. But, if we randomly choose a long enough sequence of training pairs and use a small enough step size, we can probabilistically approximate the average surface arbitrarily well.]

We define:

$$\delta_j^L \equiv \begin{cases} \varphi_j^{L\,\prime}(v_j^L)\, e_j & \text{if } L = N, \\ \varphi_j^{L\,\prime}(v_j^L) \sum_{k=1}^{K} (\delta_k^{L+1} \, w_{kj}^{L+1}), & \text{where there are } K \text{ nodes in layer } L+1, \text{ otherwise.} \end{cases}$$

[This δ is often called the local gradient for node i in layer L.]

Finally, the claim: $\frac{\partial E}{\partial w_{ji}^L} = -\delta_j^L \; y_i^{L-1}$.

[If the significance of this claim does not strike you, take a moment to notice what a computationally efficient method for computing gradient components this is. For example, if $\varphi_j^L = 1/(1 + e^{-x})$ for all j and L, then $\varphi_j^L{}'(v_j^L) = (1 - y_j^L)\, y_j^L$. Thus, computation of our gradient involves only simple arithmetic of values computed on the forward pass!]

Proof by induction on L:

Base case: Suppose L = N, and so δ_j^L is associated is with an output neuron. Then E = $\frac{1}{2}\sum_{j=1}^{j}(d_j - y_j^L)^2$, where there are J nodes in layer L, and so $\frac{\partial E}{\partial w_{ji}^L} = -(d_j - y_j^L)\ \frac{\partial y_j^L}{\partial w_{ji}^L}$ since w_{ji}^L affects only the j^{th} term of the sum. But $y_j^L = \varphi_j^L(\sum_{i=1}^{I} w_{ji}^L\, y_i^{L-1})$, where I is the number of nodes in layer L - 1, so $\frac{\partial y_j^L}{\partial w_{ji}^L} = \varphi_j^L{}'(v_j^L)\ y_i^{L-1}$, since w_{ji}^L affects only the i^{th} term of the sum. Hence, we have $\frac{\partial E}{\partial w_{ji}^L} = -(d_j - y_j^L)\varphi_j^L{}'(v_j^L)\ y_i^{L-1} = -e_j\ \varphi_j^L{}'(v_j^L)\ y_i^{L-1} = -\delta_j^L\ y_i^{L-1}$.

Induction step: Suppose L < N but the claim holds for all δ associated with nodes in layer L + 1. I.e., $\delta_j^L = \varphi_j^L{}'(v_j^L)\sum_{k=1}^{K}(\delta_k^{L+1}\, w_{kj}^{L+1})$ and each δ_k^{L+1} has the property that $\frac{\partial E}{\partial w_{kj}^{L+1}} = -\delta_k^{L+1}\, y_j^L$. We can write $\frac{\partial E}{\partial w_{ji}^L} = \frac{\partial E}{\partial y_j^L}\frac{\partial y_j^L}{\partial w_{ji}^L} = \frac{\partial y_j^L}{\partial w_{ji}^L}\frac{\partial E}{\partial y_j^L}$, and since $\partial y_j^L = \varphi_j^L(\sum_{i=1}^{I} w_{ji}^L\, y_i^{L-1})$, we have $\frac{\partial y_j^L}{\partial w_{ji}^L} = \varphi_j^L{}'(v_j^L)\ y_i^{L-1}$. From the equation $v_k^{L+1} = \sum_{j=1}^{J} w_{kj}^{L+1}\, y_j^L$, we get the following:

1) $\frac{\partial E}{\partial w_{kj}^{L+1}} = \frac{\partial E}{\partial v_k^{L+1}}\frac{\partial v_k^{L+1}}{\partial w_{kj}^{L+1}} = \frac{\partial E}{\partial v_k^{L+1}} y_j^L \implies \frac{\partial E}{\partial v_k^{L+1}} = \frac{\partial E}{\partial w_{kj}^{L+1}}\frac{1}{y_j^L}$ since $y_j^L \neq 0$,

2) $\frac{\partial E}{\partial y_j^L} = \sum_{k=1}^{K}\left(\frac{\partial E}{\partial v_k^{L+1}}\frac{\partial v_k^{L+1}}{\partial y_j^L}\right) = \sum_{k=1}^{K}\left(\frac{\partial E}{\partial v_k^{L+1}}\, w_{kj}^{L+1}\right)$.

Substituting the result of 1) into 2), we have $\frac{\partial E}{\partial y_j^L} = \sum_{k=1}^{K}\left(\frac{\partial E}{\partial w_{kj}^{L+1}}\frac{w_{kj}^{L+1}}{y_j^L}\right)$. Putting this all together, we now complete the proof:

$$\begin{aligned}
\frac{\partial E}{\partial w_{ji}^L} = \frac{\partial y_j^L}{\partial w_{ji}^L}\frac{\partial E}{\partial y_j^L} &= [\varphi_j^L{}'(v_j^L)\ y_i^{L-1}]\sum_{k=1}^{K}\left(\frac{\partial E}{\partial w_{kj}^{L+1}}\frac{w_{kj}^{L+1}}{y_j^L}\right) \\
&= [\varphi_j^L{}'(v_j^L)\ y_i^{L-1}]\sum_{k=1}^{K}\left(-\delta_k^{L+1}\, y_j^L\frac{w_{kj}^{L+1}}{y_j^L}\right) \text{ (by induction hypothesis)} \\
&= -\varphi_j^L{}'(v_j^L)\left(\sum_{k=1}^{K}\delta_k^{L+1}\, w_{kj}^{L+1}\right) y_i^{L-1} \\
&= -\delta_j^L\ y_i^{L-1}. \qquad \blacksquare
\end{aligned}$$

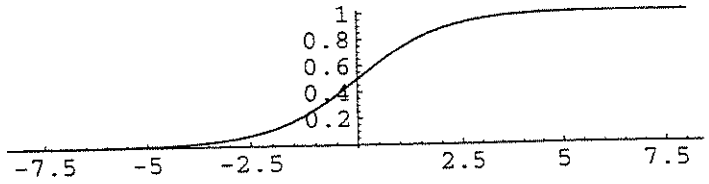
■ 2.3 Problems with Generalized Delta Rule

While the generalized delta rule drastically improves the efficiency of implementation of the steepest descent method for training artificial neural networks, there are still some serious shortcomings of the method. Naturally, these shortcomings have been studied by many people and there are now textbooks full of tricks for minimizing the nuisance of these problems. Speaking generally, the highest ambition of this REU project has been to study some of these issues from a more fundamental direction than commonly used in the relevent literature (that is,of course,a lie: the highest goal was and is to solve the problems using this approach.) This idea will be elaborated on in a later section, after the workings of networks and the problems associated with the GDR have been more thoroughly described. In this section, we will focus on the problem of convergence to local minima and on reasons the method may fail to converge altogether.

The first of these considerations, convergence to local minima, is a familiar problem from basic calculus: We are attempting to minimize a value on an unknown surface in a high-dimensional space using gradient descent. It is to be expected that we may end up converging to a local minimum. To convince ourselves that the error surface at least need not have a single global minimum with gradients at all other points in space directed towards this minimum, let's consider the following informal argument that the global minimum is not unique for most network architectures: Suppose we have a fixed architecture with more than one hidden node in a single layer, and suppose further that some pair of these nodes use the same activation function. (Notice that these hypotheses are really not very restrictive since most useful network architectures will satisfy them.) For a given set of weights and training set, we will get some value for the error of our network. If we interchange each of the weights (in-weights as well as out-weights) of our pair of nodes, it is clear that the error will remain constant, even though our weights have been altered (unless the nodes' weights were identical in the first place, in which case our architecture is not very efficient). In particular, if our error is at a minimum, we can apply this logic to see that there must be another minimum [3, pp. 194-5]. Of course, one could argue

that our minimum is a valley rather than a single point, and so there may be a path in the weight space of constant error from the first point (weights as they were before we switched them) to the second. We will see why this counter argument is usually false when we look at the XOR problem.

The second problem, the possibility of not converging at all, is a consequence of factors we can understand in a single dimension. To better understand this problem, it is necessary to look more closely at our activation functions. For clarity, let's take the specific example of $\varphi(x) = 1/(1 + e^{-x})$, whose graph is given below:



Looking at this graph, we see that the function has a relatively narrow active region. That is, only for values of x within a few units of zero is the derivative of the function much greater than zero (notice that this characteristic is common to most common choices of sigmoid.) Recall that the GDR states that if we define

$$\delta_j^L \equiv \begin{cases} \varphi_j^L{}'(v_j^L)\, e_j & \text{if } L = N, \\ \varphi_j^L{}'(v_j^L) \sum_{k=1}^{K} (\delta_k^{L+1}\, w_{kj}^{L+1}), & \text{where there are } K \text{ nodes in layer } L+1, \text{ otherwise.} \end{cases}$$

then we have $\frac{\partial E}{\partial w_{ji}^L} = -\delta_j^L\ y_i^{L-1}$. That is, the correction applied to a weight is directly proportional to the value of the sigmoid associated with the node to which the weight connects. If the sigmoid's argument is out of the active range, then corrections to the weight will be small (and changes in the error will be very small.) This condition is referred to *premature saturation* [3, pp. 157-8.] This problem is reduced somewhat by employing the common convention of initializing the weights to small random numbers, but taking this measure does not eliminate the problem completely.

The most common way of compensating for these small gradient values is to increase the step size in the training algorithm (there are various methods for doing this; the most prominent method is a technique called *momentum*.) The problem with increasing the step size, however, is that if the weight value corrections are resulting in sigmoid arguments which move towards the active region of the sigmoid, then once we get to the active region our gradient value will immediately increase. Our large step size and gradient may now cause the weight correction to push us back out of the active region, possibly resulting in a net gain of zero for the entire process. On the other hand, keeping the step size small usually results in the slow convergence generally considered characteristic of steepest descent methods.

In situations which have problems analogous to the above, the principle behind our (John's, originally) "highest goal idea" has been applied with dramatic results. These results and this analogy are strong motivations behind this project. Again, the idea will be discussed shortly.

■ Section 3: The XOR Problem

▪ 3.1 Problem Definition and Implementation of a Simple ANN

The XOR problem has a special place in the history of neural networks, and is therefore touched on in most introductory ANN textbooks (for more on this history, see [1].) Our initial attraction to the problem and the value of discussing it here are derived from the simplicity of the XOR problem, which lends itself to implementations which are of comprehensible scale. In this sense, the problem is helpful for developing intuition and understanding of neural networks. XOR is an abbreviation of "exclusive or." Logically, (A XOR B)=(A OR B) AND NOT(A AND B).

My XOR ANN borrowed the architecture in Figure 2 of WordDiagrams.doc from [1] (the version in the book had a *bias* in the output neuron; I ommitted this) and used a steepest-descent

algorithm to minimize the error. The top input component is 1 for all input vectors. The role its weights fulfill is called *bias* for the hidden layer nodes; the necessity of bias in this network will be made clear in section 3.2. For this simple architecture, it is not difficult to write an explicit expression for the network's output in terms of its input vector, so in my code (see Code Appendix A) this expression is used directly, as is an explicit formula for the gradient.

The program has an outer loop which sets the weights to random initial values and then enters an inner loop which performs a number of iterations of steepest descent. When the inner loop is exited the program gives an output of: total error over all training pairs and a list of weights if the error is small, or only the total error if the error is not small. A typical output is:

2.4999
2.
1.50124
2.
2.

In this case, each of the errors is not "small." The code uses four training vectors corresponding to each of thefour boolean possibilities for the XOR problem, (0, 0), (0, 1), (1, 0), (1, 1). These error values raise an interesting question: Since the network's output sigmoid has a value of either one or zero for almost all arguments, how can the error of the network be a multiple of 0.5 rather than one? It turns out that this sample of errors is not a fluke--on most runs of the program, the errors are multiples of 0.5 (run program A to see this.) The next section explains this phenomenon.

▪ 3.2 Disection of the XOR Problem

Most of the work in this section was motivated from the anomalous error values mentioned at the end of section 3.1. Notice that the output for the network described in section 3.1 is given by: $F(x_1, x_2) = \varphi[w^2_{11}\varphi_1(w^1_{10} + w^1_{11}x_1 + w^1_{12}x_2) + w^2_{12}\varphi_2(w^1_{20} + w^1_{21}x_1 + w^1_{22}x_2)]$. For most values of **x** and **w**, $\varphi_1(w^1_{10} + w^1_{11}x_1 + w^1_{12}x_2)$ will be very close to one or zero, and likewise, $\varphi_2(w^1_{20} + w^1_{21}x_1 + w^1_{22}x_2)$ will

be approximately one or zero, so the final output of the network will usually be (approximately):

$\varphi[w^2_{11} + w^2_{12}]$ if $(w^1_{10} + w^1_{11}x_1 + w^1_{12}x_2) > 0$ and $(w^1_{20} + w^1_{21}x_1 + w^1_{22}x_2) > 0$;

$\varphi[w^2_{11}]$ if $(w^1_{10} + w^1_{11}x_1 + w^1_{12}x_2) > 0$ and $(w^1_{20} + w^1_{21}x_1 + w^1_{22}x_2) < 0$;

$\varphi[w^2_{12}]$ if $(w^1_{10} + w^1_{11}x_1 + w^1_{12}x_2) < 0$ and $(w^1_{20} + w^1_{21}x_1 + w^1_{22}x_2) > 0$;

$\varphi[0]$ if $(w^1_{10} + w^1_{11}x_1 + w^1_{12}x_2) < 0$ and $(w^1_{20} + w^1_{21}x_1 + w^1_{22}x_2) < 0$;

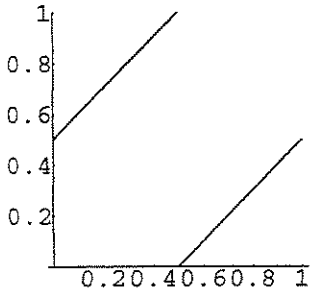
By reasoning similar to the above, we can see that the first three of these expressions will usually have values very near one or zero, but $\varphi[0] = 0.5$. Thus, we typically have error values which are sums of ones, zeros, and 0.5's.

We've answered the question at hand, but we can still do more in the way of truly understanding the dynamics of our network. Let's consider the network weights fixed and take a closer look at the four most likely network outputs given above. If we think of an input vector **x** as a *point* in the $x_1 \times x_2$ plane and think of the expressions L_1, $w^1_{10} + w^1_{11}x_1 + w^1_{12}x_2 = 0$, and L_2, $w^1_{20} + w^1_{21}x_1 + w^1_{22}x_2 = 0$, as *lines* in the $x_1 \times x_2$ plane, then $\varphi_1(\mathbf{x})$ returns a one if **x** is above L_1, and a zero if it is below L_1. Likewise, φ_2 can be thought of as asking whether **x** is above L_2. Knowing this, we can choose L_1 and L_2 as we wish, thus dividing up our input space anyway we choose. [In 3.1, the explanation of the necessity of the bias terms was deferred to this section. Their importance should be clear now (a bias is the negative of the y-intercept of a line). Also, the claim made in 2.3, that the error surface minimum is not likely to be a valley, is substantiated by the above discussion.] However, there is still a problem with our network. For **x** below both L_1 and L_2 we have $\varphi_1(\mathbf{x})$ $\varphi_2(\mathbf{x})$ 0, so our network will have to output a 0.5 in the lower region in which we want an output of one or zero (depending on our exact specification of the desired XOR output.) It turns out, then, that the bias to the output neuron which I chose to omit was actually necessary.

■ 3.3 Using Our Insights on the XOR Problem

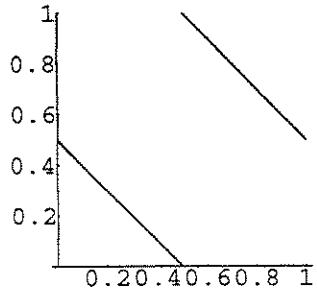
In light of the previous section, to design an ANN for the XOR problem we must first decide how we want to place the lines which divide up our input space. Somehow, we need to insure that the points (1,0) and (0,1) are in areas which output a one, and that (1,1) and (0,0) are in spaces which output zero. It is worth noticing that in order to do this, it is clear that at least two lines (and hence two hidden layer nodes) are needed; such limitations on the capabilities of particular fixed architectures are central to much of the study of ANNs. Once the lines are chosen, we can simply assign weight values accordingly and the network is finished.

My first effort at this approach the XOR problem assumed the following input space partition was desireable:



[For a 3D picture of the output surface, run program B.1, but beware: the output uses about 115kB]

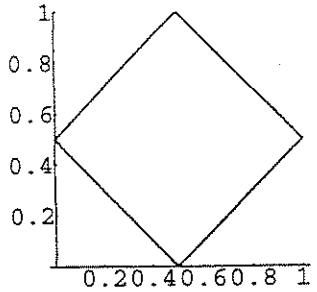
However, it turns out that the design in [1] assumes a different partition:



[B.2 will provide a better picture for this.]

This discrepency reminds us that the output for the middle region is really arbitrary, and so we may

want to consrtuct a network which outputs one for areas near (1, 0) or (0, 1), zero for areas around (1, 1) or (0, 0), and .5 (analogous to a "maybe" answer) for the middle region:



[You guessed it: B3...]

Obvoiusly, we will need four hidden nodes if we are to accomplish this partition of the error surface. We will therefore use the architecture in Figure 3 of WordDiagrams.doc.

By inspection, we can see that one choice of lines is

$L_1 = \{(x_1,x_2)\colon\ x_2 + x_1 - 0.5 = 0\}$

$L_2 = \{(x_1,x_2)\colon\ x_2 - x_1 + 0.5 = 0\}$

$L_3 = \{(x_1,x_2)\colon\ x_2 + x_1 - 1.5 = 0\}$

$L_4 = \{(x_1,x_2)\colon\ x_2 - x_1 - 0.5 = 0\}$

The first layer of weights can now be simply read off. If we let F_i be the function which gives L_i for i = 1, 2, 3, 4, then we can express the output of our network as $F(x_1, x_2) = \varphi[w^2_{11}\varphi_1\{F_1(x_1) - x_2\} + w^2_{12}\varphi_2\{F_2(x_1) - x_2\} + w^2_{13}\varphi_3\{F_3(x_1) - x_2\} + w^2_{14}\varphi_4\{F_4(x_1) - x_2\}]$. Now, we must go back to our chosen output specifications to determine the values for the w^2_{1i}. We want a one if we are above L_1, L_2, and L_4, but below L_3 *or* if we are above L_1, but below L_2, L_3, and L_4. And we want to return a zero if we are above L_1, L_3, and L_4, but below L_2 *or* if we are above L_4, but below L_1, L_2, and L_3. Finally, we want to return 0.5 if we are above L_1 and L_4, but below L_2, and L_3. Thus, we want our w^2_{1i} to satisfy the linear system:

$w^2_{11} + w^2_{12} + w^2_{14} > 0,$

$w_{11}^2 > 0,$

$w_{11}^2 + w_{13}^2 + w_{14}^2 < 0,$

$w_{14}^2 < 0,$

$w_{11}^2 + w_{14}^2 = 0.$

One set of weights which satisfies the linear system is $w_{11}^2 = w_{12}^2 = 1$, $w_{13}^2 = w_{14}^2 = -1$. Program B.3 uses the network derived above to plot the output surface.

■ Section 4: Alternative Gradients

▪ 4.1: Rethinking the Gradient

In Section 4, we will work up to the theory behind this project's "highest goal," which has been alluded to in previous chapters. First, let's take another look at the steepest descent method. Suppose we have a quantity $\phi(\mathbf{x})$ that we wish to minimize. Recall that when we use steepest descent we tacitly assume that the standard Euclidean gradient gives the negative of the "best" direction in which to choose successive $\mathbf{x}$ values for the purpose of minimizing $\phi(\mathbf{x})$. This is a perfectly natural assumption since we each have a lifetime of experiences reinforcing our intuition for the Euclidean notion of distance which is the basis of this gradient. However, there are other (non-Euclidean) ways of defining a metric on a vector space which result in gradient definitions which can be perfectly valid. Furthermore, there are examples (we will look at one) where the gradient resulting from an intelligently chosen metric results in a gradient descent which drastically outperforms its Euclidean counterpart. Our ambition, therefore, is to make some headway in determining a metric whose properties are particularly well-suited for finding a minimum on the types of error surfaces encountered in ANN training.

Of course, it is necessary to define some terms for this discussion to be meaningful. A real-valued function defined on a vector space X is called a *norm* (denoted $\| \cdot \|$) if the following

properties hold for all $\mathbf{x}$, $\mathbf{y}$ in X and all scalars α :

i. $\| \mathbf{x} \| = 0$ iff $\mathbf{x} = 0$,

ii. $\| \mathbf{x} + \mathbf{y} \| \leq \| \mathbf{x} \| + \| \mathbf{y} \|$,

iii. $\| \alpha\mathbf{x} \| = |\alpha| \cdot \| \mathbf{x} \|$ [5].

A normed vector space becomes a *metric space* if we define a metric ρ by $\rho(\mathbf{x}, \mathbf{y}) = \|\mathbf{x} - \mathbf{y}\|$, where $\| \cdot \|$ satisfies the criteria of the above definition [5]. A metric space is *complete* if every Cauchy Sequence in the space converge [5]. If a normed vector space is complete in a metric, then we call the space a *Banach Space* [5]. Finally, we call a Banach space H a *Hilbert Space* if there is a function (called the *inner product*) $\langle \mathbf{x}, \mathbf{y} \rangle : H \times H \longrightarrow \mathbb{R}$ with the properties:

i. $\langle \alpha_1 \mathbf{x}_1 + \alpha_2 \mathbf{x}_2 , \mathbf{y} \rangle = \alpha_1 \langle \mathbf{x}_1, \mathbf{y} \rangle + \alpha_2 \langle \mathbf{x}_2, \mathbf{y} \rangle$,

ii. $\langle \mathbf{x}, \mathbf{y} \rangle = \langle \mathbf{y}, \mathbf{x} \rangle$,

iii. $\langle \mathbf{x}, \mathbf{x} \rangle = \| \mathbf{x} \|^2$ [5].

These properties are easily verified for the Euclidean innner product, $\langle \mathbf{x}, \mathbf{y} \rangle = x_1 y_1 + x_2 y_2 + \ldots + x_n y_n$.

Finally, we consider a function $\phi : \mathbb{R}^n \longrightarrow \mathbb{R}$ and we look at some posssible definitions of the gradient of ϕ, $\nabla\phi$. Typically, we use the definition $\nabla\phi(\mathbf{x}) = [\frac{\partial\phi}{\partial x_1}(\mathbf{x}), \frac{\partial\phi}{\partial x_2}(\mathbf{x}), \ldots , \frac{\partial\phi}{\partial x_n}(\mathbf{x})]$, but an equivalent definition is: $\nabla\phi(\mathbf{x})$ is the element of $\mathbb{R}^n$ which satisfies the equation $\phi'(\mathbf{x})\mathbf{h} = \langle \mathbf{h}, \nabla\phi(\mathbf{x}) \rangle$ for all $\mathbf{h} \in \mathbb{R}^n$ [4]. Any inner product on the linear space $\mathbb{R}^n$ can written in the form $\langle A\mathbf{x}, \mathbf{y} \rangle$ where A is some positive definate symmetric matrix and $\langle \cdot \rangle$ is the Euclidean inner product [4]. We will use the notation $\langle \mathbf{x}, \mathbf{y} \rangle_A = \langle A\mathbf{x}, \mathbf{y} \rangle$, and we now pursue the question of how various inner products (i.e., various choices of A) affect the gradient given by the second definition above. As posed in [4], the question is: "Given $\mathbf{x} \in \mathbb{R}^n$, what member $\nabla_A\phi(\mathbf{x})$ of $\mathbb{R}^n$ gives us the identity: $\phi'(\mathbf{x})\mathbf{h} = \langle \mathbf{h}, (\nabla_A\phi)(\mathbf{x}) \rangle_A$ for all $\mathbf{h} \in \mathbb{R}^n$?" Suppose we have an object, $\nabla_A\phi(\mathbf{x})$, which gives us this identity. Then we have $\phi'(\mathbf{x})\mathbf{h} = \langle \mathbf{h}, (\nabla_A\phi)(\mathbf{x}) \rangle_A = \langle A\mathbf{h}, (\nabla_A\phi)(\mathbf{x}) \rangle = \langle \mathbf{h}, A^T(\nabla_A\phi)(\mathbf{x}) \rangle = \langle \mathbf{h}, A(\nabla_A\phi)(\mathbf{x}) \rangle$, so $A(\nabla_A\phi)(\mathbf{x}) = \nabla$

$\phi(\mathbf{x})$. or $(\nabla_A\phi)(\mathbf{x}) = A^{-1}(\nabla\phi)(\mathbf{x})$ (since A positive definate $\Longrightarrow$ A invertible) [4].

Since the generality of the above discussion may leave the reader unconvinced about the meaning (and hence usefulness) of an alternative gradient, before moving on to an example of the application of such a gradient, we will show that $(\nabla_A\phi)(\mathbf{x})$ is at least a descent direction. The essentials of the following arguement are from [4]. If we define $f(\delta) = \phi[\mathbf{x} - \delta(\nabla_A\phi)(\mathbf{x})]$ then we have $f'(0) = -\phi'(\mathbf{x})[(\nabla_A\phi)(\mathbf{x})]$ (by the chain rule)

$= - \langle (\nabla_A\phi)(\mathbf{x}), \nabla\phi(\mathbf{x}) \rangle$ (by our gradient definition)

$= - \langle A^{-1}(\nabla_A\phi)(\mathbf{x}), \nabla\phi(\mathbf{x}) \rangle_A$ (by definition of $\langle \cdot \rangle_A$)

$= - \langle (\nabla_A\phi)(\mathbf{x}), A^{-1}\nabla\phi(\mathbf{x}) \rangle_A$

$= - \langle (\nabla_A\phi)(\mathbf{x}), (\nabla_A\phi)(\mathbf{x}) \rangle_A$

$= \| (\nabla_A\phi)(\mathbf{x}) \|_A^2 < 0$ whenever $(\nabla_A\phi)(\mathbf{x}) \neq 0$. ■

■ 4.2: Example of Using an Alternative Gradient Effectively

In this section, we will outline the effects of a non-Euclidean gradient for a steepest descent method of solving a simple differential equation, y'=y on the interval [0, 1]. The basis for this discussion is Chapter 2 of [4]. Rather than approaching this problem from the usual continuous function perspective, we will *discretize* the function y. That is, we will represent y as a vector, $\mathbf{x} = (x_0, x_1, \ldots, x_n)$. We now want to design a function $\phi_n : \mathbb{R}^{n+1} \longrightarrow \mathbb{R}^n$ such that a $\phi_n(\mathbf{x}) = 0 \Longrightarrow \mathbf{x}$ represents a function y which satisfies y' = y. We choose $\phi_n(\mathbf{x}) = \sum_{i=1}^{n} \left(\frac{x_i - x_{i-1}}{\gamma_n} - \frac{x_i + x_{i-1}}{2} \right)^2$, where $\gamma_n = 1/n$. We now pick for our initial guess some $y \in C^{(3)}$ such that at least one of the inequalities y'(0) -y(0) 0 or y'(1) - y(1) 0 holds. We now define a sequence of points $\{w^n\}_{n=1}$ which are representative of y in that $w_i^n = y(i/n)$ for i = 1, 2, ..., n.

In [4], it is shown that as n increases, the usefulness of iterations of Euclidean gradient descent

deteriorates quite badly. In fact, $\lim_{n\to} \frac{\phi_n(w^n - \delta_n(\phi_n \nabla)(w^n))}{(\phi_n(w^n)} = 1$, even when δ_n is chosen optimally for the purpose of minimizing $\phi_n(w^n - \delta_n \nabla \phi_n(w^n))$. In other words, in the limit case, Euclidean gradient descent is completely futile. Suppose, on the other hand, we use a gradient based on the $H^{1,2}$ inner product. For two differentiable functions x(t), y(t), the $H^{1,2}$ inner product of x and y is given by $\int_{t=0}^{1}(x\cdot y + x'\cdot y')dt$, so for our discretized version, we use an analogous version which yeilds the $\|\mathbf{x}\|_A = \sum_{i=1}^{n}[(\frac{x_i - x_{i-1}}{\gamma_n})^2 + \frac{x_i + x_{i-1}}{2})^2]^{1/2}$ [4]. (Notice that we could write out the matrix A explicitly.) In this case, it is shown in [4] that for optimally chosen δ_n, $\frac{\phi_n(w^n - \delta_n(\phi_n \nabla_A)(w^n))}{(\phi_n(w^n)} \leq \frac{1}{9}$ for n = 1, 2, ...

I should point out that the above discussion is really not so detached from practical implementation as one might suspect at first reading. In real programmed implementations of the above processes, it has been found that, in order to reach a particular level of accuracy, the Euclidean method requires up to 500,000 iterations for n = 100, whereas the $H^{1,2}$ version reaches the desired accuracy after only 7 iterations [4].

■ 4.3: The Task Ahead of Us

At this point, we have an idea of some of the benefits of using specialized gradients in steepest descent algorithms. We also have a basic understanding of the workings of layered feed-forward ANNs. We have seen some of the reasons for the difficulties encountered in actual implementation of steepest descent methods using these gradients. If we could somehow apply the underlying principles of Sections 4.2 and 4.3 to the problematic error surfaces of ANNs, then we might see a dramatic improvement in convergence time for the training of ANNs. This would be no small breakthrough: the difficulty of training the weights in ANNs is one of the biggest obstacles in the way of the application of ANNs to truly large-scale problems [1].

Unfortunately, there are no general procedures for determining which inner products will lead to particularly effective gradients for traversing a space for a given problem. We therefore need to study

further some of the dynamics of ANN training, as well as the characteristics of effectively trained ANN weight configurations. Hopefully, we may be able to ferret out some principles at work in the training process which may point the way to more effective gradients. It is with this in mind that we look to a new task which will require a neural network of a larger scale than that of the XOR problem.

■ Section 5: Digit Recognition

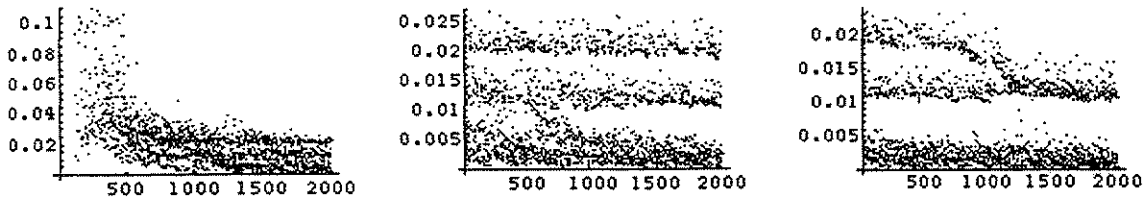
▪ 5.1 Why We Are Interested in Applying an ANN to this Problem

In the digit recognition problem, the task, simply put, is to design and train a neural network to recognize each of the digits 0..9. This problem, of course, has a large practical significane and neural nets seem to be particularly well-suited for automating hand-written character recognition. One reason we chose to implement this problem is that the exact demands placed on a character-recognition system can be varied quite widely (some use several thousand neurons [1], whereas our version used only 50.) Hence, we can implement a real character recognition ANN on a scale which can be handled by *Mathematica*® in a reasonable amount of time, and so we can take advantage of *Mathematica's*® graphics capabilities and many built-in functions. These features are valuable tools both in developing basic understanding of ANNs and in attempting to gain deeper insight into the dynamics of training an ANN. At this point in the project, we are seeking insights into aspects of network training which may hold the key to applying an alternative gradient version of steepest descent to network training.

▪ 5.2 Implementation of Standard Back-Propagation

Our network uses a fully-connected 30×10×10 feed-forward ANN. That is, it has thirty input nodes, ten hidden nodes, and ten output nodes with each node in the hidden layer recieving input from all input nodes, and each output node recieving input from all hidden nodes. The training set was

designed with a 5-pixel × 6-pixel input screen in mind. We concatenated the rows of this abstract screen to make input vectors of length thirty. Each desired output vector is ten units long and has all entries set equal to 0.1, except for the i^{th} entry (where the corresponding input vector is a representation of the digit *i)* which is set to 0.9. Our program initializes all weights in the network to small random numbers and performs a steepest-descent traversal of the error surface using back-propagation (each iteration uses a single randomly chosen training pair.) The output is a graph which plots the error as a function of iteration. A typical output is below:



(To run this program, run program C.2, but read the paragraph in C.1 first.) For the output above, the step size was quite large, 7, but we still saw definate convergence. In fact, the error gets quite small after the first 500-1000 iterations. This does not mean that the problems of slow convergence and failure to converge are not valid concerns in most networks, but rather that this particular network (with the sigmoid using $a = 1$) does not tend to overshoot its target values irrevocably. In fact, if we increase the parameter a,, we can see that progress in training slows dramatically.

▪ 5.3 Overtraining

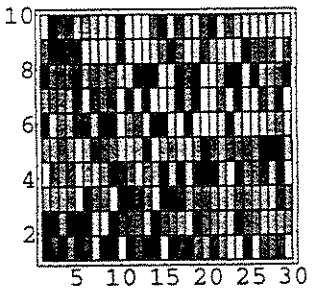
The ability of a network to learn using a finite set of training vectors and to effectively generalize what it has learned to arbitrary cases of a problem is one of the most valuable aspects of

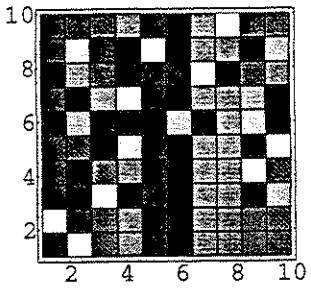
ANN applications. It is sometimes found, however, that networks can *overtrain*, or learn a training set so thouroughly that generalization performance suffers. The code in C.3 contains a training set and a separate "test set" of input/desired output vectors. The code was written to investigate overtraining in a network, so the output plots the network's error on its training set versus the error on the test set as training progresses. Occasionally, the output will include a small region in which an inverse relationship appears, but on the whole, the relationship appears to be fairly direct. Due to this lack of evidence of overtraining tendencies for the network and the time constraints of an eight-week program, we have not looked into this aspect of training at length. The reader is invited to run (and modify) the code in C.3 to investigate this princple further.

▪ 5.4 A More Direct Approach to the Search for Promising Norms

Recall that the effective gradient for solving the problem $y' - y = 0$ was based on the $H^{1,2}$ inner product, $\int_{t=0}^{1}(x \cdot y + x' \cdot y')dt$. In this inner product space, two functions (or vectors if we discretize) are considered close if and only if they are both pointwise close *and* have similar derivatives. This fact is somewhat satisfying since the a solution to the problem must have very specific properties both pointwise and in terms of the derivative. In this sense, one perspective we might hope to find particularly helpful for choosing an effective gradient for a problem is to try to determine various characteristics which will be important to any solution. Once critical characteristics of solutions are identified, it may be possible to design an inner product which "measures" these characteristics, giving its gradient an "insider's" perspective on the error surface. With this thought in mind, I set out to display the values of the weights of a trained network graphically, in hopes of determining meaningful tendencies among solutions. If important trends are to be found among the weights, I was unable to decipher them. Of course, there are many possibilities for display designs and I have not looked at them

all. One such display, produced by the code in C.4, is given below. The first graph displays the weights to the hidden layer of the network, and the second shows weights to the output layer.





■ 6: Conclusion

It is a little out of place to write a conclusion to a project which still has many interesting questions open and viable paths unexplored. In answer to this dilemma, I have included at the end of this report a "Miscellaneous" section which mentions some of the ideas I might look into if I had unlimited time. However, I feel I should make some comments on the many concepts we have been able to explore during this program. While we have not been able to make any breakthroughs for the field of ANNs, we have gained a fairly thorough understanding of many fundamental elements of the field. I personally have learned about some relatively advanced topics in analysis and differential equations, and have gained a respectable proficiency in *Mathematica* programming. A more fundamental benefit I have recieved from this program has been in working with the translation of powerful, abstract mathematical concepts into practical, concrete applications. This is something

particularly valuable to me in that my mathematical tastes are decidedly abstract, but I still place great importance on seeing more tangible results of my work.

■ Miscellaneous Comments

In this informal section, I will discuss a few things I don't want to forget, and which the reader may find interesting or helpful.

At the end of Section 2.3, I mentioned that the problem of saturation is usually dealt with by adjusting the step size. It occurs to me that there is another approach which attacks the problem in a more fundamental way, and so might give better results. Recall that we were looking at the sigmoid $\varphi(x) = 1/(1 + e^{-ax})$ with $a = 1$, and that the source of the problem is that this function has a narrow active region. By taking smaller values of a, we can stretch the active region all we want. The idea, then, might be to start training the network with, say, $a = .001$ and to gradually increase a as the training progresses. Of course, the step size might need to be defined in terms of a in order to take full advantage of this idea.

Another idea relating to stepsize involves the second derivative. If we could come up with a clever way to bound this, then we could have the stepsize depend on this bound: a bounded second derivative means a limit to the curvature of the surface, which means the that we could insure that we don't overshoot our minimum. I actually spent a couple of days looking into this with no earth-shattering results. For more on uses of second derivatives in training ANNs, see [1] or [3].

In Section 5.4, we looked directly at the weight values of trained networks to try to gain insight into characteristics of successful weight systems. An even more direct method might be to look directly at successful learning rules. On very simple networks, evolutionary programming has been been used with some modest success to *evolve* learning rules for networks [6]. This has not (that I know of) produced more useful methods than were already known, but it does allow one to see the levels of

efficiency of various training methods. I don't know how practical this idea is, but it may be a good way to find enough different training methods to allow one to to identify trends among the more successful learning rules. It would be up to the designer of the evolutionary system to arrange things so that one could determine the matrix A implicit in any given learning rule. One simple possibility might be to have entries of the matrix A be elements of the evolving string (chromosome) in the genetic algorithm. For greater efficiency, it might be beneficial to have a single value in the chromosome representing several matrix entries (a diagonal band, for instance, may be a worthwhile choice since diagonally banded matrices are useful for applying many differential operators.) For more on evolutionary computing, see [6].

■ Bibliography

[1] Simon Haykin. *Neural Networks: A Comprehensive Foundation*, Macmillan College Publishing Company, New York, 1994.

[2] James A. Freeman. *Simulating Neural Networks with Mathematica®*, Addison-Wesley Publishing Company, Inc., Reading, Massachusetts, 1994.

[3] Robert A. Schalkoff. *Artificial Neural Networks*, McGraw-Hill, New York, 1997.

[4] J. W. Neuberger. *Sobolev Gradients and Differential Equations*, Springer-Verlag, Berlin, 1997.

[5] H. L. Royden. *Real Analysis (Third Edition,)* MacMillan Publishing Company, New York, 1988.

[6] Melanie Mitchell. *An Introduction to Genetic Algorithms,* MIT Press, Cambridge, Massachusetts, 1996.

■ Code Appendix

■ A: Steepest descent for the XOR Problem

```
Clear[ w111, w112, w113, w121, w122, w123, w211, w212, sigV1, sigV2, sigPV1, sigPV2, yminusd, err];

step = .3;

For [j = 1, j <= 5, j += 1,
(********** Begin Outer Loop **********)

SeedRandom[];
w111 = Random[Real, {-5, 5}];
w112 = Random[Real, {-5, 5}];
w113 = Random[Real, {-5, 5}];
w121 = Random[Real, {-5, 5}];
w122 = Random[Real, {-5, 5}];
w123 = Random[Real, {-5, 5}];
w211 = Random[Real, {-5, 5}];
w212 = Random[Real, {-5, 5}];
weights = {w111, w112, w113, w121, w122, w123, w211, w212};

a = 10;

sig[x_] = 1/(1 + E^(-a*x));

sigP[x_] = a*E^(-a*x)*(1 + E^(-a*x))^(-2);

trainIter[vecX_] :=
 Module[{V1, V2, V3, sigV1, sigV2, sigPV1, sigPV2, yminusd, temp},
        V1 = vecX[[1]]*w111+vecX[[2]]*w112+vecX[[3]]*w113;
        V2 = vecX[[1]]*w121+vecX[[2]]*w122+vecX[[3]]*w123;
        sigV1 = sig[V1];
        sigV2 = sig[V2];
        sigPV1 = sigP[V1];
        sigPV2 = sigP[V2];
        yminusd = sig[w211*sigV1 + w212*sigV2]-vecX[[4]];
        temp = {0, 0, 0, 0, 0, 0, 0, 0};
        temp[[1]] = sigP[w211*sigV1 + w212*sigV2] * w211 * sigPV1 * vecX[[1]];
        temp[[2]] = sigP[w211*sigV1 + w212*sigV2] * w211 * sigPV1 * vecX[[2]];
        temp[[3]] = sigP[w211*sigV1 + w212*sigV2] * w211 * sigPV1 * vecX[[3]];
        temp[[4]] = sigP[w211*sigV1 + w212*sigV2] * w212 * sigPV2 * vecX[[1]];
        temp[[5]] = sigP[w211*sigV1 + w212*sigV2] * w212 * sigPV2 * vecX[[2]];
        temp[[6]] = sigP[w211*sigV1 + w212*sigV2] * w212 * sigPV2 * vecX[[3]];
        temp[[7]] = sigP[w211*sigV1 + w212*sigV2] * sigV1;
        temp[[8]] = sigP[w211*sigV1 + w212*sigV2] * sigV2;
        temp *= yminusd;
  Return[{temp, yminusd}]; ];

 For     [k = 1, k <= 50, k += 1,
 (*========== Begin Inner Loop ==========*)

 err = 0;
 graderr = {0, 0, 0, 0, 0, 0, 0, 0};
```

```
{tmpgraderr, tmperr} = trainIter[{1, 0, 0, 0}];
graderr += tmpgraderr;
err += Abs[tmperr];
{tmpgraderr, tmperr} = trainIter[{1, 0, 1, 1}];
graderr += tmpgraderr;
err += Abs[tmperr];
{tmpgraderr, tmperr} = trainIter[{1, 1, 0, 1}];
graderr += tmpgraderr;
err += Abs[tmperr];
{tmpgraderr, tmperr} = trainIter[{1, 1, 1, 0}];
graderr += tmpgraderr;
err += Abs[tmperr];

weights -= step*graderr;
{w111, w112, w113, w121, w122, w123, w211, w212} = weights;

(*================== End Inner Loop ================================*)
];
If[err < .1, Print[weights, "    ", err], Print[err] ];

(******************** End Outer Loop ******************************)
];
```

■ B: Error surfaces for XOR

■ B.1: My version

```
In[5]:=
        a = 10;          (* Try varying a  *)
        sig[x_] = 1/(1 + E^(-a*x));

        z[x_, y_] = sig[ + sig[y- x- .5] - sig[y - x +.5] + .5 ];
        Plot3D[z[x, y],{x, 0, 1},{y, 0, 1}, PlotPoints->30];
```

■ B.2: Version in [1]

```
a = 10; (* Try varying a  *)
sig[x_] = 1/(1 + E^(-a*x));


z[x_, y_] = sig[sig[y + x -.5] - sig[y + x -1.5] - .5];
Plot3D[z[x, y],{x, 0, 1},{y, 0, 1}, PlotPoints->30]
```

■ B.3: Alternative XOR output

```
a = 10;          (* Try varying a  *)
sig[x_] = 1/(1 + E^(-a*x));


z[x_, y_] = sig[ sig[y+x-.5] + sig[y-x-.5] - sig[y+x-1.5] - sig[y-x+.5]];
Plot3D[z[x, y],{x, 0, 1},{y, 0, 1}, PlotPoints->30]
```

■ C: Character recognition back-propagation

■ C.1: Read this section before using program C.2 or C.3.

Programs C.2 and C.3 require lengthy input sections which I do not want to print for practical reasons. I have therefore divided the cells containing these programs so that the input cell can be closed. To run a program, simply highlight the cells containing the program and its input and merge these cells. The program should be ready to run.

C.2: Back propagation with error graph output

```
(*  This program plots the network's performance
  on the training vectors vs. perfomance on certian test vectors.  I
  used it to investigate the notion of 'overtraining' a network.    *)

(* bpnStandard
  is from:          p. 81 of Freeman's Simulating Neural Nets with Mathematica. [4] *)



ClearAll[sigmoid, bpnStandard, inNumber, hidNumber, outNumber, ioPairs, eta, numIters,
  errors, hidWts, outWts, ioP, inputs, hidweights,
 outweights, outDesired, hidOuts, outputs, outErrors, outDelta, a, hidDelta];

a = 1;

sigmoid[x_] := N[ 1 / (1 + E^(-x)) ];


bpnStandard[ hidWates_, outWates_, inNumber_,
  hidNumber_, outNumber_, ioPairs_, tstPairs_, eta_, numIters_] :=

 Module[{errors, ioP, inputs,  hidWts = hidWates, outWts = outWates, outDesired, hidOuts,
      outputs, outErrors, outDelta, hidDelta, i},

     errors = Table[

       (* select ioPair *)
       k = Random[Integer, {1, Length[ioPairs]}];
       ioP = ioPairs[[k]];
       inputs = ioP[[1]];
       outDesired = ioP[[2]];

       (* forward pass  *)
       hidOuts = sigmoid[hidWts . inputs] ;
       outputs = sigmoid[outWts . hidOuts] ;

       (* determine errors, deltas  *)
       outErrors = outDesired - outputs ;
       outDelta =  a outErrors ( outputs ( 1 - outputs)) ;
       hidDelta =  a (hidOuts (1 - hidOuts)) Transpose[outWts] . outDelta ;

       (* update weights  *)
       outWts += eta Outer[Times, outDelta, hidOuts];
       hidWts += eta Outer[Times, hidDelta, inputs];

       (* enter squared errors into table  *)
       outErrors . outErrors, {numIters}];  (*  end Table  *)

       Return[{hidWts, outWts, errors}];
  ]; (* end of module  *)
```

```
SeedRandom[];
inNumber = 30;
hidNumber = 10;
outNumber = 10;

hidweights = Table[ Table[ Random[ Real, {-0.1, 0.1}], {inNumber}], {hidNumber}];
   (*  essentially, a [hidNumber] X [inNumber] matrix   *)

outweights = Table[ Table[ Random[ Real, {-0.1, 0.1}], {hidNumber}], {outNumber}];
   (*  an [outNumber] X [hidNumber] matrix              *)

{hidweights, outweights, errorTbl} = bpnStandard[hidweights,
  outweights, inNumber, hidNumber, outNumber, ioPairs, testPairs, 7, 2000];
 ListPlot[errorTbl];

{hidweights, outweights, errorTbl} = bpnStandard[hidweights,
  outweights, inNumber, hidNumber, outNumber, ioPairs, testPairs, 7, 2000];
 ListPlot[errorTbl];

{hidweights, outweights, errorTbl} = bpnStandard[hidweights,
  outweights, inNumber, hidNumber, outNumber, ioPairs, testPairs, 7, 2000];
   ListPlot[errorTbl];
```

■ C.3: Overtraining The network.

```
(*  This program plots the network's performance
  on the training vectors vs. perfomance on certian test vectors.  I
  used it to investigate the notion of 'overtraining' a network.   *)

(* bpnStandard
  is from:          p. 81 of Freeman's Simulating Neural Nets with Mathematica.  *)



ClearAll[sigmoid, bpnStandard, inNumber, hidNumber, outNumber, ioPairs, eta, numIters,
 errors, hidWts, outWts, ioP, inputs, hidweights,
 outweights, outDesired, hidOuts, outputs, outErrors, outDelta, hidDelta];

sigmoid[x_] := N[ 1 / (1 + E^(-x)) ];

errPair[ testPairs_, trnPairs_, hidweights_, outweights_]  :=
  Module[{tstErr, trnErr, k, inp, hdotps, otp, current, otpDesired},
   tstErr = 0;
   For[ k = 1, k <= Length[testPairs], k++,
     current = testPairs[[k]];
     inp = current[[1]];
     otpDesired = current[[2]];
     (* forward pass  *)
     hdotps = sigmoid[hidweights . inp] ;
     otp = sigmoid[outweights . hdotps] ;
     tstErr += N[ (otpDesired - otp) . (otpDesired - otp) ];
     ];
   trnErr = 0;
   For[ k = 1, k <= Length[trnPairs], k++,
     current = trnPairs[[k]];
     inp = current[[1]];
     otpDesired = current[[2]];
     (* forward pass  *)
     hdotps = sigmoid[hidweights . inp] ;
     otp = sigmoid[outweights . hdotps] ;
     trnErr += N[ (otpDesired - otp) . (otpDesired - otp) ];
     ];
   Return[{trnErr, tstErr}];
   ];

bpnStandard[ hidWates_, outWates_, inNumber_,
  hidNumber_, outNumber_, ioPairs_, tstPairs_, eta_, numIters_] :=

 Module[{errors, ioP, inputs,  hidWts = hidWates, outWts = outWates, outDesired, hidOuts,
      outputs, outErrors, outDelta, hidDelta, i},

     errors = Table[0, { Floor[numIters / 100] }];
     For[ i = 1, i <= numIters, i++,

       (* select ioPair *)
       k = Random[Integer, {1, Length[ioPairs]}];
       ioP = ioPairs[[k]];
       inputs = ioP[[1]];
```

```
      outDesired = ioP[[2]];

      (* forward pass  *)
      hidOuts = sigmoid[hidWts . inputs] ;
      outputs = sigmoid[outWts . hidOuts] ;

      (* determine errors, deltas  *)
      outErrors = outDesired - outputs ;
      outDelta = outErrors ( outputs ( 1 - outputs)) ;
      hidDelta = (hidOuts (1 - hidOuts)) Transpose[outWts] . outDelta ;

      (* update weights  *)
      outWts += eta Outer[Times, outDelta, hidOuts];
      hidWts += eta Outer[Times, hidDelta, inputs];

     (* enter squared errors into table  *)
    If[ Mod[i, 100] == 0,
        errors[[i / 100]] = errPair[ tstPairs, ioPairs, hidWts, outWts] ];
      ];
      Return[{hidWts, outWts, errors}];
]; (* end of module  *)


SeedRandom[];
inNumber = 30;
hidNumber = 10;
outNumber = 10;

hidweights = Table[ Table[ Random[ Real, {-0.1, 0.1}], {inNumber}], {hidNumber}];
   (*  essentially, a [hidNumber] X [inNumber] matrix   *)

outweights = Table[ Table[ Random[ Real, {-0.1, 0.1}], {hidNumber}], {outNumber}];
   (*  an [outNumber] X [hidNumber] matrix              *)

{hidweights, outweights, errorTbl} = bpnStandard[hidweights,
  outweights, inNumber, hidNumber, outNumber, ioPairs, testPairs, 2, 3000];
 ListPlot[errorTbl];

{hidweights, outweights, errorTbl} = bpnStandard[hidweights,
  outweights, inNumber, hidNumber, outNumber, ioPairs, testPairs, 2, 3000];
 ListPlot[errorTbl];

{hidweights, outweights, errorTbl} = bpnStandard[hidweights,
  outweights, inNumber, hidNumber, outNumber, ioPairs, testPairs, 2, 3000];
   ListPlot[errorTbl];
```

■ C.4: Display of Weights

```
(* This program should be run AFTER the backpropagation program
   has been run to converge the weigts.  *)

w = hidweights;
v = outweights;
For[
      ioP = ioPairs[[k]];
      inputs = ioP[[1]];
      outDesired = ioP[[2]];
    (* forward pass  *)
      hidOuts = sigmoid[hidWts . inputs] ;
      outputs = sigmoid[outWts . hidOuts] ;
    (* determine errors, deltas  *)
      outErrors = outDesired - outputs ;
      outDelta = outErrors ( outputs ( 1 - outputs)) ;
      hidDelta = (hidOuts (1 - hidOuts)) Transpose[outWts] . outDelta ;
    (* update weights  *)
      outWts += eta Outer[Times, outDelta, hidOuts];
      hidWts += eta Outer[Times, hidDelta, inputs];
    (* enter squared error into table  *)
      outErrors . outErrors , {numIters}];

DensityPlot [ w[[Round[y]]][[Round[x]]] ,
 {x, 1, inNumber}, {y, 1, hidNumber}, PlotPoints -> {inNumber, hidNumber}];
DensityPlot [Abs[ w[[Round[y]]][[Round[x]]] ],
 {x, 1, inNumber}, {y, 1, hidNumber}, PlotPoints -> {inNumber, hidNumber}];

DensityPlot [ v[[Round[y]]][[Round[x]]] ,
 {x, 1, hidNumber}, {y, 1, outNumber}, PlotPoints -> {hidNumber, outNumber}];
DensityPlot [Abs[ v[[Round[y]]][[Round[x]]] ],
 {x, 1, hidNumber}, {y, 1, outNumber}, PlotPoints -> {hidNumber, outNumber}];

DensityPlot [
 0, {x, 1, inNumber}, {y, 1, hidNumber}, PlotPoints -> {inNumber, hidNumber}];
```